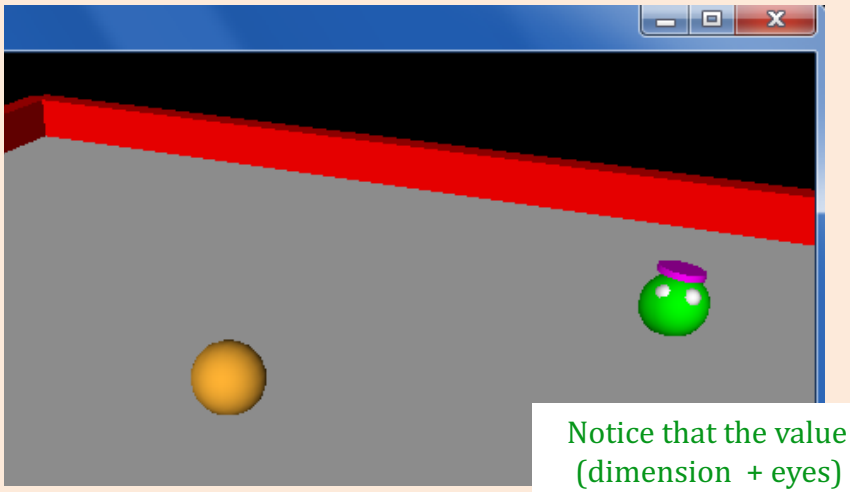
Any memorable prose composed for (well, by) your program yesterday?

Today: 3d Graphics



Notice that the value of (dimension + eyes) is conserved!



Three-eyed? Today, we're ***3d***'ed!



Python features, *motivated* by VPython

Tuples are similar to lists, but they're parenthesized:

```
T = (4,2)      V = (1,0,0)
```

Example of a two-element ***tuple*** named T and a three-element tuple named V

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of ***default arguments*** in a function definition

Tuples!

but what about that *name*?!



Lists that use parentheses are called ***tuples***:

```
>>> T = (4, 2)
>>> T
(4, 2)

>>> T[0]
4

>>> T[0] = 42
Error!

>>> T = ('a',2,'z')
>>>
```

Tuples are immutable lists: you can't change their elements...

...but you can always redefine the whole variable, if you want!

+ Tuples are more memory & time efficient
+ Tuples ***can*** be dictionary keys: ***lists can't be***
- ***But you can't change tuples' elements...***

Tuples!

but what about that *name*?!



Lists that use parentheses are called ***tuples***:

```
>>> T = (4, 2)
>>> T
(4, 2)

>>> T[0]
```

Etymology [edit]

The term originated as an abstraction of the sequence: single, double, triple, quadruple, quintuple, sextuple, septuple, octuple, ..., n-tuple, ..., where the prefixes are taken from the Latin names of the numerals. The unique 0-tuple is called the null tuple.

...but you can always redefine the whole variable, if you want!

```
>>> T = ('a',2,'z')
>>>
```

+ Tuples are more memory & time efficient
+ Tuples ***can*** be dictionary keys: ***lists can't be***
- ***But you can't change tuples' elements...***

Tuple problems…

Creating 0- and 1-tuples would seem like a problem!



How do you create a tuple of 0 or 1 elements?

```
>>> T = (4)
>>> T
4

>>> T = (4,)
(4,)

>>> T = ()
()

>>> T = 1, 2, 3, 'a'
>>> (1, 2, 3, 'a')
```

Python details... (that VPython uses)

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of ***default*** arguments for x and y

```
f(x = 4, y = 2)
```

example of ***named*** arguments for x and y

Function arguments *look like* tuples,
but they're not quite the same…

Python details in VPython…

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of an ordinary function call—totally OK

```
f(4, 2)
```


example of ***default arguments***

```
f()
```


example using only one ***default argument***

```
f(1)
```


example of a ***named argument***

```
f(y = 1)
```

Python details in VPython…

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of an ordinary function call—totally OK

```
f(4, 2)
```



```
42
```

example of ***default arguments***

```
f()
```


example using only one ***default argument***

```
f(1)
```


example of a ***named argument***

```
f(y = 1)
```

Python details in VPython…

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of an ordinary function call—totally OK

```
f(4, 2)
```



```
42
```

example of ***default arguments***

```
f()
```



```
47
```

example using only one ***default argument***

```
f(1)
```


example of a ***named argument***

```
f(y = 1)
```

Python details in VPython…

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of an ordinary function call—totally OK

```
f(4, 2)
```



```
42
```

example of ***default arguments***

```
f()
```



```
47
```

example using only one ***default argument***

```
f(1)
```



```
27
```

example of a ***named argument***

```
f(y = 1)
```

Python details in VPython…

Functions can have ***default arguments*** and can take ***named arguments***

```
def f(x = 3, y = 17):
    return 10*x + y
```

example of an ordinary function call—totally OK

```
f(4, 2)
```



```
42
```

example of ***default arguments***

```
f()
```



```
47
```

example using only one ***default argument***

```
f(1)
```



```
27
```

example of a ***named argument***

```
f(y = 1)
```



```
31
```

Named arguments

```
def f(x = 2, y = 11):
    return x + 3*y
```

Try These!

`f(3, 1)`

`f()`

`f(3)`

`f(y = 2, x = 1)`

What will these function calls to **`f`** return?

None of the above are 42!

What call to **`f`** returns the string **'Lalalalala'** ?

These are tuples—they work like lists!

What is **`f( ( ), (1,0) )`** ?

What is the ***shortest*** call to **`f`** returning 42?

it's only four characters, too!

Extra… what does this return? **x = -6; y = 60; f(y = x, x = y)**

Named arguments

```
def f(x = 2, y = 11):
    return x + 3*y
```

Try These!

`f(3, 1)` ➡ `6`

`f()` ➡ `35`

`f(3)` ➡ `36`

`f(y = 2, x = 1)` ➡ `7`

What will these function calls to `f` return?

None of the above are 42!

What call to `f` returns the string **'Lalalalala'** ?

These are tuples—they work like lists!

What is `f( ( ), (1,0) )` ?

What is the ***shortest*** call to `f` returning 42?

it's only four characters, too!

Extra… what does this return? **x = -6; y = 60; f(y = x, x = y)**

Named arguments

```
def f(x = 2, y = 11):
    return x + 3*y
```

Try These!

`f(3, 1)` ➡ `6`

`f()` ➡ `35`

`f(3)` ➡ `36`

`f(y = 2, x = 1)` ➡ `7`

What will these function calls to `f` return?

None of the above are 42!

What call to `f` returns the string **'Lalalalala'**? ➡ `f('La','la')`

These are tuples—they work like lists!

What is `f( ( ), (1,0) )`? ➡ `(1,0,1,0,1,0)`

What is the ***shortest*** call to `f` returning 42? ➡ `f(9)`

it's only four characters, too!

Extra… what does this return? **x = -6; y = 60; f(y = x, x = y)** ➡

Named arguments

```
def f(x = 2, y = 11):
    return x + 3*y
```

Try These!

`f(3, 1)` ➡ `6`

`f()` ➡ `35`

`f(3)` ➡ `36`

`f(y = 2, x = 1)` ➡ `7`

What will these function calls to `f` return?

None of the above are 42!

What call to `f` returns the string **'Lalalalala'**? ➡ `f('La','la')`

These are tuples—they work like lists!

What is `f( ( ), (1,0) )`? ➡ `(1,0,1,0,1,0)`

What is the ***shortest*** call to `f` returning 42? ➡ `f(9)`

it's only four characters, too!

Extra… what does this return? **x = -6; y = 60; f(y = x, x = y)** ➡ **42**

VPython

www.vpython.org

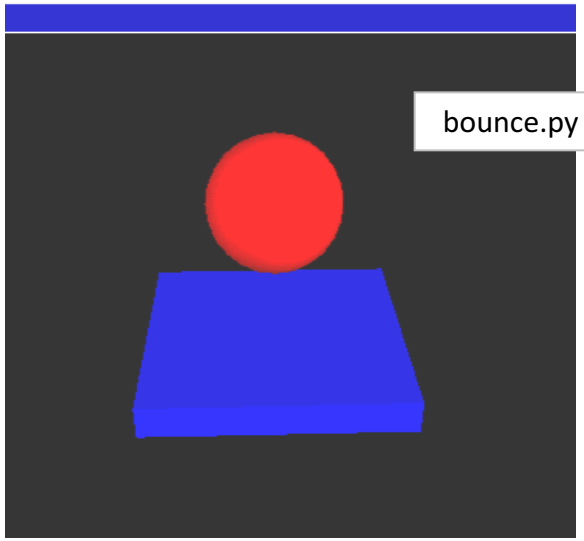
Where were we… ?

VPython
3D Programming for Ordinary Mortals



stonehenge.py

Built *by* and *for* physicists to simplify 3d simulations



bounce.py

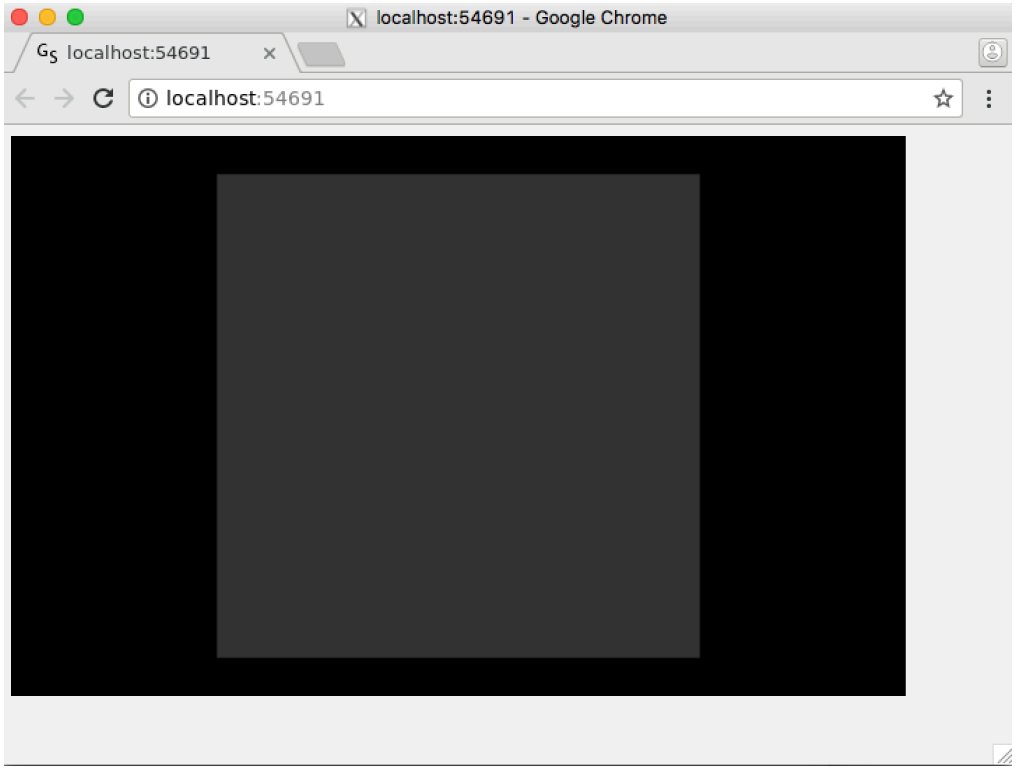
Lots of available classes, objects and methods in its **API**

Available at glowscript.org

Installing VPython

```
$ pip3 install vpython
```

```
>>> from vpython import *
>>> b = box()
```



Getting Help

http://vpython.org/
http://glowscript.org/

vpython vs. glowscript?

vpython runs locally on your machine
glowscript runs in a browser

Both share the same API

API

... stands for *Application Programming Interface*

A ***programming*** description of how to use a software library

A demo of vpython's API:

```
from vpython import *
def main():
    c = cylinder()
    while True:
        rate(10)

# This calls main when the file is run
if __name__ == "__main__":
    main()
```

What's **`cylinder`**?

What's **`c`**?

What's **`main`**?

What are those last two lines?

VPython example API calls

API

... stands for *Application Programming Interface*

www.glowscript.org/docs/VPythonDocs/index.html

Home

If you're new to Python and VPython: Introduction

A VPython tutorial

Pictures of 3D objects

Choose a 3D object

Work with 3D objects

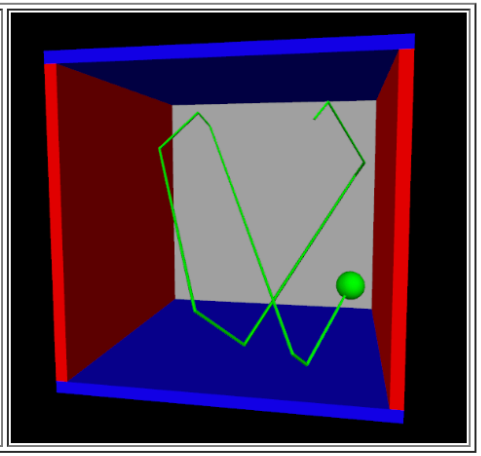
Canvases/Events

What's new

Classic VPython web site
VPython license
Python web site

VPython

GlowScript VPython and VPython 7



Using VPython to create 3D animations

VPython makes it unusually easy to write programs that generate navigable real-time 3D animations. It is based on the Python programming language which is widely used in introductory programming courses thanks to its clean design, and it is also widely used in science and business.

Classic VPython was originated by David Scherer in 2000. In 2011 David Scherer and Bruce Sherwood initiated the development of GlowScript, a similar programming environment but which runs in a browser. In 2014 it became possible to use RapydScript, a programming language very similar to Python, to support VPython programs in the GlowScript environment.

Here is an **overview** of the project, and here is a **log of developments**. Here is the **open source** for this project. There is also **technical documentation** about the GlowScript environment, including how to use JavaScript or RapydScript for your programs.

There is also VPython 7 originated by John Coady in 2014 and under continuing development by John Coady, Ruth Chabay, Bruce Sherwood, Steve Spicklemire, and Matt Craig. It uses GlowScript VPython syntax but with standard Python, thereby providing access to standard Python modules. For details see **vpython.org**. Also of interest is a discussion about plans for the **future evolution** of VPython.

This documentation describes both GlowScript VPython and VPython 7.

API ... stands for *Application Programming Interface*

www.glowscript.org/docs/VPythonDocs/arrow.html

Home

If you're new to Python and VPython: Introduction

A VPython tutorial

Pictures of 3D objects

Choose a 3D object

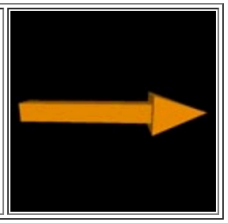
Work with 3D objects

Canvases/Events

What's new

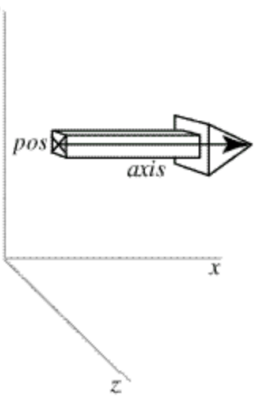
Classic VPython web site
VPython license
Python web site

arrow



The arrow object has a straight box-shaped shaft with an arrowhead at one end. The following statement will display an arrow pointing parallel to the x axis:

```
pointer = arrow(pos=vector(0,2,1),
      axis=vector(5,0,0), shaftwidth=1)
```



The arrow object has the following attributes and default values, like those for cylinders: pos (0,0,0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), opacity (1), shininess (0.6), emissive (False), **texture**, and up (0,1,0). As with box, the up attribute is significant for arrow because the shaft and head have square cross sections, and setting the up attribute rotates the arrow about its axis. Additional arrow attributes:

shaftwidth By default, shaftwidth = 0.1*(length of arrow)

headwidth By default, headwidth = 2*shaftwidth

headlength By default, headlength = 3*shaftwidth

Assigning shaftwidth = 0 makes it be the default (0.1 times the length of the arrow). If you don't explicitly set headwidth or headlength, its length is based as shown above on shaftwidth (either the default or whatever value you specify for shaftwidth). If headlength becomes larger than half the length of the arrow, or the shaft becomes thinner than 1/50 the length, the entire arrow is scaled accordingly.

This default behavior makes the widths of very short arrows shrink, and the widths of very long arrows grow (while displaying the correct total length). If you prefer that shaftwidth and headwidth not change as the arrow gets very short or very long, specify the desired shaftwidth. In this case the only adjustment that is made is that headlength is adjusted so that it never gets longer than half the total length, so that the total length of the arrow is correct. This means that very short, thick arrows look similar to a thumbtack, with a nearly flat head.

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a box, sphere, or ring it corresponds to the center of the object.

If you include make_trail=True when you create the object, a trail will be left behind the object as you move it. For related options, see **Leaving a Trail**.

See **Rotating an Object** for an easy way to change the orientation of an object.

See description of Additional Attributes available for all 3D display objects.

VPython

Set up world with 3d objects…

```
b = box()
b.color = color.red
# 1: change colors…
```

color.red or a tuple

```
# 3: add objects
scene.autoscale = False
c = cylinder(pos = vec(4,0,0))
a = sphere(pos = vec(0,0,4))
```

more objects and API calls

(most will be added ***before*** the while loop)

Then, run a simulation using those objects…

```
while True:
    rate(30)   # times/sec
    print("b.pos is", b.pos)
    # 2: change position
```

```
# 4: define velocity  (above loop)
# 5: then, change pos  (in loop)
```

```
if __name__ == "__main__":
  main()
```

vectors

b.pos, **b.vel**,... are vectors

```
b.vel = vector(1,  0,  0)
b.pos = vector(0,  0,  0)

b.pos = b.pos + b.vel*0.2
```

vel.x

vel.y

vel.z

named components

scalar multiplication

component-by-component addition

let's compare with tuples...

vectors

act like "*arrows*"

The vector Object

The vector object is not a displayable object but is a powerful aid to 3D computations.

```
vector(x,y,z)
```

Returns a vector object with the given components, which are made to be floating-point (that is, 3 is converted to 3.0).

Vectors can be added or subtracted from each other, or multiplied by an ordinary number. For example,

```
v1 = vector(1,2,3)
v2 = vector(10,20,30)
print(v1+v2) # displays <1 22 33>
print(2*v1)  # displays <2 4 6>
```

You can refer to individual components of a vector:

v2.x is 10, v2.y is 20, v2.z is 30

It is okay to make a vector from a vector: vector(v2) is still vector(10,20,30).

The form vector(10,12) is shorthand for vector(10,12,0).

A vector is a Python sequence, so v2.x is the same as v2[0], v2.y is the same as v2[1], and v2.z is the same as v2[2].

Vector functions

The following functions are available for working with vectors:

http://www.glowscript.org/docs/VPythonDocs/vector.html

vectors

lots of support… (don't write your own)

The vector Object

The vector object is not a displayable object but is a powerful aid to 3D computations.

Vector functions

The following functions are available for working with vectors:

mag(A) = A.mag = |A|, the magnitude of a vector

mag2(A) = A.mag2 = |A|*|A|, the vector's magnitude squared

norm(A) = A.norm() = A/|A|, a unit vector in the direction of the vector

dot(A,B = A.dot(B) = A dot B, the scalar dot product between two vectors

cross(A,B) = A.cross(B), the vector cross product between two vectors

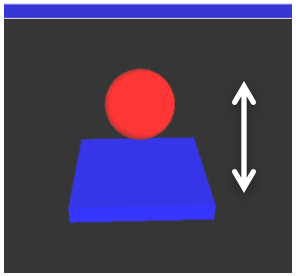
proj(A,B) = A.proj(B) = dot(A,norm(B))*norm(B), the vector projection of A along B

comp(A,B) = A.comp(B) = dot(A,norm(B)), the scalar projection of A along B

diff_angle(A,B) = A.diff_angle(B), the angle between two vectors, in radians

rotate(A,theta,B) = A.rotate(theta,B) = rotate(vector=A, angle=theta, axis=B), result of rotating A through theta around B

http://www.glowscript.org/docs/VPythonDocs/vector.html



vPython!

Look over this VPython program to determine

(1) How many vPython classes are used in this code? ____

(2) How many vPython objects are used here? _____

(3) What line of code handles ***collisions*** ?

(4) How does ***physics*** work? Where is it?

(5) Tricky! How many tuples are here?

```
from vpython import *

def main():
    """docstring!"""
    floor = box(length=4,width=4,height=0.5,color=color.blue)
    ball = sphere(pos=vector(0,8,0),radius=1,color=color.red)
    ball.vel = vector(0,-1,0)
    RATE = 30
    dt = 1.0/RATE

    while True:
        rate(RATE)
        ball.pos += ball.vel*dt

        if ball.pos.y < ball.radius:
            ball.vel.y *= -1.0
        else:
            ball.vel.y += -9.8*dt
```

what is this += doing?

what is this if/else doing?

Let's run it!

Lab goals

(0) Try out VPython!

(1) Make a few changes to the starter code

(2) Implement ***walls*** and ***wall collisions***…

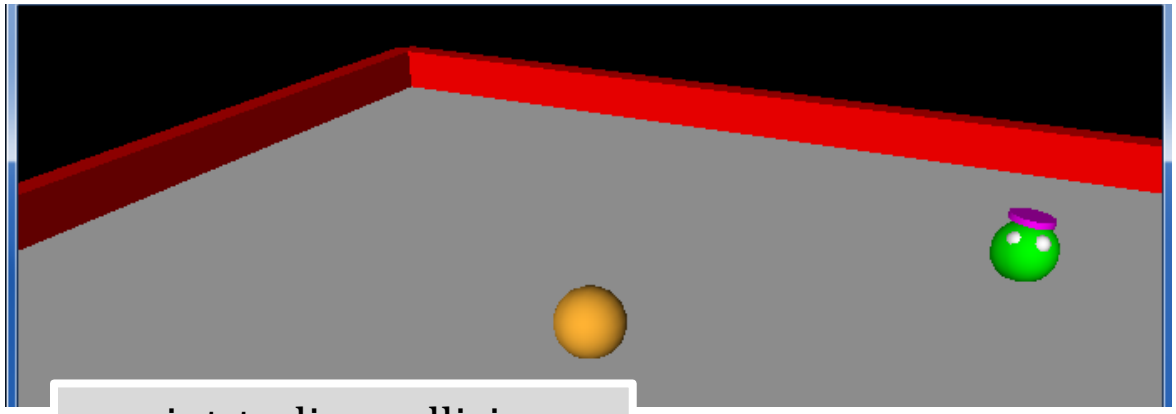
(3) Improve your alien!

(4) *Optional*: add scoring, enemies, or a moving target, hoops, traps, holes, etc.—***your own game***…



final-project option

Collisions...



point-to-line collisions

```
# if the ball hits wallA
if ball.pos.z < wallA.pos.z:        # hit - check for z
    ball.pos.z = wallA.pos.z        # bring back into bounds
    ball.vel.z *= -1.0              # reverse the z velocity

# if the ball hits wallB
if ball.pos.x < wallB.pos.x:        # hit - check for x
    ball.pos.x = wallB.pos.x        # bring back into bounds
    ball.vel.x *= -1.0              # reverse the x velocity

# if the ball collides with the alien, give a vertical velocity
if mag(ball.pos - alien.pos) < 1.0:
    print("To infinity and beyond!")
    alien.vel = vector(0, 1, 0)
```

point-to-point collisions

compound

Home Pictures of 3D objects

Choose a 3D object ▾ Work with 3D objects ▾ Canvases/Events ▾



The **compound** object lets you group objects together and manage them as though they were one object, by specifying in the usual way pos, color, size (and length, width, height), axis, up, opacity, shininess, emissive, and texture. Moreover, the display of a complicated compound object is faster than displaying the individual objects one at a time. (In GlowScript version 2.1 the **details were somewhat different**.)

The object shown above is a compound of a cylinder and a box:

```
handle = cylinder( size=vec(1,.2,.2),
                   color=vec(0.72,0.42,0) )

head = box( size=vec(.2,.6,.2),
            pos=vec(1.1,0,0),
            color=color.gray(.6) )

hammer = compound([handle, head])
hammer.axis = vec(1,1,0)
```

The size of the object: After creating the compound named "hammer", **hammer.size** represents the size of the bounding box of the object.

compound

Home Pictures of 3D objects

Choose a 3D object ▾ | Work with 3D objects ▾ | Canvases/Events ▾

compound



The **compound** object lets you group objects together and manage them as though they were one object, by specifying in the usual way pos, color, size (and length, width, height), axis, up, opacity, shininess, emissive, and texture. Moreover, the display of a complicated compound object is faster than displaying the individual objects one at a time. (In GlowScript version 2.1 the **details were somewhat different**.)

The object shown above is a compound of a cylinder and a box:



What's what here?

```
alien_body = sphere( size=1.0*vector(1,1,1), pos=vector(0,0,0), color=color.green )
alien_eye1 = sphere( size=0.3*vector(1,1,1), pos=.42*vector(.7,.5,.2), color=color.white )
alien_eye2 = sphere( size=0.3*vector(1,1,1), pos=.42*vector(.2,.5,.7), color=color.white )
alien_hat = cylinder( pos=0.42*vector(0,.9,-.2), axis=vector(.02,.2,-.02),
                                                 size=vector(0.2,0.7,0.7), color=color.magenta)
alien_objects = [alien_body, alien_eye1, alien_eye2, alien_hat]

com_alien = compound( alien_objects, pos=starting_position )
```

```
# +++ start of EVENT_HANDLING section -- separate functions
#                                  for keypresses and mouse clicks...

def keydown_fun(event):
    """Function called with each key pressed."""
    ball.color = randcolor()
    key = event.key
    ri = randint( 0, 10 )
    print("key:", key, ri)  # prints the key pressed

    amt = 0.42   # "strength" of the keypress's velocity changes
    if key == 'left':
        ball.vel = ball.vel + vector(0, 0, -amt)
    elif key == 'down':
        ball.vel = ball.vel + vector(-amt, 0, 0)
    elif key == 'right':
        ball.vel = ball.vel + vector(0, 0, amt)
    elif key == 'up':
        ball.vel = ball.vel + vector(amt ,0, 0)
    elif key in " rR":
        ball.vel = vector(0, 0, 0) # reset! via the spacebar
        ball.pos = vector(0, 0, 0)
```

Random change of the sphere's color

Printing is great for debugging!

Variables make it easy to change behavior across many lines of code (Here, all four motion directions)

Have shortcuts to make your game easier—or reset it!

key presses...